Online Feedback-Directed Optimization of Java

Matthew Arnold[†‡] Michael Hind[‡] Barbara G. Ryder[†]

[†]Rutgers University, Piscataway, NJ, 08854
[‡]IBM T.J. Watson Research Center, Hawthorne, NY, 10532
{marnold,ryder}@cs.rutgers.edu, hind@watson.ibm.com

ABSTRACT

This paper describes the implementation of an online feedback-directed optimization system. The system is fully automatic; it requires no prior (offline) profiling run. It uses a previously developed low-overhead instrumentation sampling framework to collect control flow graph edge profiles. This profile information is used to drive several traditional optimizations, as well as a novel algorithm for performing feedback-directed control flow graph node splitting. We empirically evaluate this system and demonstrate improvements in peak performance of up to 17% while keeping overhead low, with no individual execution being degraded by more than 2% because of instrumentation.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*runtime environment, compilers, optimization*

General Terms

Algorithms, Experimentation, Measurement, Languages

Keywords

Dynamic optimization, adaptive optimization, online algorithms, virtual machines

1. INTRODUCTION

Many of today's Java™ virtual machines (JVMs) employ an optimizing compiler to improve application performance. Because such optimization is performed at runtime, attention has focused on limiting the overhead of runtime optimization by either reducing the execution time of the optimizer or applying optimization only to key portions of the application [30, 39, 43, 21, 6]. The latter selective optimization approach requires a careful tradeoff between the benefit of reduced optimization time and the increased runtime cost of detecting these key portions.

OOPSLA'02, November 4–8, 2002, Seattle, Washington, USA.

Although challenging to achieve, runtime optimization also offers an opportunity: the ability to tailor optimizations to the current execution environment. Such an approach, typically referred to as feedback-directed optimization, not only instructs the optimizer what to optimize, but also specifies how the method should be optimized based on the behavior of the application. Several systems [26, 35, 34] have shown that performance can be improved substantially by exploiting invariant runtime values; however, these systems were not fully automatic and relied on programmer directives to identify regions of code to be optimized. Many researchers (ex., [37, 40, 31, 18]) have focused on performing feedback-directed optimizations in which the application behavior is captured from a prior execution of the program because these profiling executions typically incur significant overhead. Although the resulting performance improvements are often promising, this approach fails in scenarios where 1) it is impractical to collect a profile prior to execution, or 2) the application's behavior differs from its behavior during the profiling run.

This work describes our experience with the implementation of an *online* feedback-directed optimization system. The system is fully automatic; it requires no prior profiling run. Performing profiling and optimizations online is an attractive approach because it avoids the drawbacks of offline profiling. However, such an approach does incur runtime overhead to collect the profile information, make decisions based on the resulting profile, and perform the actual feedback-directed optimizations. These three steps incur overhead and thus create the potential for *degrading* performance rather than improving it.

Our implementation is an instance of the architecture of the Jikes™ RVM[1] adaptive optimization system [6], which provides a high-level design for feedback-directed optimizations. Further, we utilize the general *instrumentation sampling* framework [9] to collect control flow graph edge profiles. This framework significantly reduces instrumentation overhead without significantly degrading profiling quality.

The contributions of this paper are

- **Implementation** We describe how the general adaptive optimization architecture described in [6] can be instantiated to perform online feedback-directed optimizations. Specifically, we describe 1) how edge profiles are collected using the instrumentation sampling

[1]The Jikes RVM is an open-source research virtual machine for the Java programming language that was formerly called Jalapeño [1]. It is available at www.ibm.com/developerworks/oss/jikesrvm.

framework [9]; 2) how to enhance the cost/benefit model given in [6] to perform feedback-directed optimizations; and 3) how to perform four feedback-directed optimizations, including a novel algorithm for control flow graph node splitting.

- **Empirical Evaluation** We empirically evaluate our implementation in the Jikes RVM, a high-performing system that has competitive performance with state-of-the-art JVMs. We demonstrate that our online approach can improve the performance of long-running versions of the SPECjvm98 benchmarks by up to 16.9% with an mean improvement of 4.3%, without degrading the performance of short-running programs. We also show improved throughput for the SPECjbb2000 server benchmark by up to 3.5%.

The remainder of the paper is organized as follows. Section 2 describes the background for this work, including summaries of the Jikes RVM adaptive optimization system and the instrumentation sampling framework. Section 3 describes the implementation of our system. Section 4 describes the empirical evaluation of our system. Section 5 discusses related work and Section 6 presents our conclusions.

2. BACKGROUND

This section provides the background for this work. It describes the challenges for performing online feedback-directed optimizations (FDO), previous approaches to solving these problems, the instrumentation sampling framework we utilize to achieve efficient profiling, and the adaptive optimization system that provides the context for our implementation.

2.1 Challenges in Performing Online FDO

A system that performs feedback-directed optimizations (either offline or online) strives to improve performance by using profiling information to identify common execution patterns that can be exploited when performing optimizations. However, online feedback-directed optimization is more difficult than offline feedback-directed optimization for two main reasons, increased runtime overhead and the accuracy and availability of the profile information.

Overhead Several sources of overhead exist in performing feedback-directed optimizations online. For example, there is overhead for 1) collecting the profiling information, 2) examining the profile data and making optimization decisions, and 3) performing the actual feedback-directed optimizations. Care must be taken during these steps or performance may be reduced rather than improved.

Profile accuracy and availability When using offline profile information, the profile is usually assumed to 1) incur no runtime cost because it is performed during a prior profiling run, 2) be accurate, and 3) be available prior to execution. With online profiling, none of these assumptions are true. By definition the profile is collected at runtime of the performance run. Further, the accuracy and availability of profile data are dependent on i) what is profiled, ii) when the profiling is performed, and iii) how long the profiling is performed.

There is no simple solution for these problems that is best in all scenarios; solving them involves balancing a number of tradeoffs. For example, profiling for a short duration reduces profiling overhead and allows the profile to be available sooner, but may result in a less accurate profile. Similarly, profiling soon after the program begins execution enables earlier availability of the profile information, but may come at the cost of profile accuracy when the program's startup behavior is not representative of its long-running behavior.

Another difference between offline and online profiling is that offline profiles typically represent an aggregate of the entire application's execution. Optimizations using such a profile use the application's average characteristics, but ignore application phases. In contrast, online profiles can represent the execution only up to the current time. Optimizations using such a profile must rely on some prediction heuristic about future application behavior.

2.2 Existing Online Strategies

Systems that use online profiling information to drive feedback-directed optimizations generally employ one or more of the following three strategies to collect their profiles: profile early during unoptimized execution, profile during optimized execution, or profile throughout the entire execution.

Profile early during unoptimized execution Hölzle and Ungar [30] pioneered the use of adaptive recompilation and online feedback directed optimization in the Self-93 system. The goal of their system was to avoid long optimization pauses in interactive applications by focusing optimization efforts on the performance-critical methods of the application. Counters were placed on method entries of unoptimized methods and the resulting profile was used to make decisions regarding when to optimize, and what methods to optimize. The method entry counts were also used to perform feedback-directed receiver class prediction and method inlining.

The Hotspot [39] and the IBM DK 1.3.0 [43] JVMs collect finer-grained information about each method during interpretation, before the method has been optimized. Hotspot profiles never-null object references and receiver types at call sites, while the IBM DK 1.3.0 profiles branch frequencies. To reduce overhead, no profiling occurs during execution of the optimized code.[2] The overhead of this approach is likely to be low relative to the already poor performance of the unoptimized (or interpreted) code. Another advantage is that the profile information is available when the method is first recompiled, enabling feedback-directed optimizations to occur as early as possible.

However, there are several limitations to profiling unoptimized code. First, methods are profiled only during their early stages of execution, which can be ineffective, or even counter-productive, for a method whose dominant behavior is not exercised during its early execution. To identify rarely executed code for driving partial method compilation Whaley [49] proposes a three-stage execution model in which fine-grained profiling is avoided during interpretation, but is inserted in the second stage, where lightweight optimizations are performed. Although this approach may help in some cases, it is not a general solution because no mecha-

[2]The IBM DK 1.3.0 [43] does perform instrumentation of optimized code (as discussed later in this section); however, the branch frequencies are profiled during instrumentation only.

nism is provided for collecting a profile after a fixed initial duration; applications may vary in the length of their initialization code or may undergo phase shifts.

A second limitation of profiling unoptimized code is that it makes certain profiling information more difficult to collect. For example, optimizing compilers often perform aggressive method inlining, which can drastically change the structure of a method. Additional optimizations are then typically performed on the resulting inlined code. However, determining the hot paths through the inlined code can be nontrivial when using a profile obtained prior to inlining, thereby limiting the availability of accurate profile information to the subsequent optimizations.

Profile optimized code Profiling optimized code can address the shortcomings described above, but suffers from a new problem: the overhead of profiling can be substantial, particularly when instrumentation is inserted into the program to collect fine-grained profiling information. Because of this overhead, few systems currently use this approach. To keep overhead low, the IBM DK 1.3.0 [43, 44] uses *code patching* to dynamically remove instrumentation after it has executed for a fixed number of times. The overhead and scalability of code patching is a concern, particularly when collecting profiles that would require substantial code patching. For example, recording all memory references [19] or applying several different types of instrumentation together at once could be difficult to perform effectively using a patching-only approach. In these scenarios substantial overhead could occur during the profiling period, and execution may remain partially degraded even after patching occurs.

To avoid significant overhead, profiles are collected in short bursts. Although this approach reduces profiling overhead, it also increases the chances of collecting a non-representative profile, particularly if the program behavior varies over time. Finally, code patching may introduce architecture-specific complexities such as maintaining cache consistency.

In some cases instrumentation is simply applied directly. Kistler [33] describes using online instrumentation in an Oberon system, where the instrumentation is inserted with little concern for its overhead. The system computes how long the program needs to execute to recover instrumentation overhead time. For applications that do not run long enough, performance can be severely degraded.

Exception Directed Optimization (EDO) [38] also adds instrumentation directly to exception handling to profile exception paths from a `throw` statement to the catching handler. However, the overhead of this instrumentation is likely to appear lightweight given the infrequent occurrence and relative expense of exception handling.

Sample throughout execution Some types of profile information can be collected by means other than code instrumentation. For example, the Jikes RVM's adaptive system [6] profiles the call graph by periodically sampling the call stack at regular time intervals. A similar approach is used in the IBM DK 1.3.0 [44], where an instrumenting profiler dynamically generates instrumentation code for a specific method. Unfortunately, many types of fine-grained profiling information, such as basic-block frequencies and value profiles, can be difficult to collect using this approach. Current systems that collect this types of profiles via sampling require relatively complex infrastructure as well as hardware support [4, 15, 41].

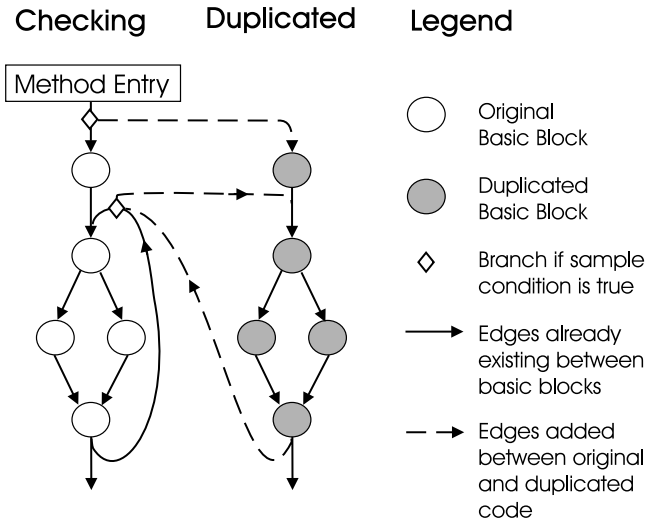

Figure 1: Overview of instrumentation sampling framework [9]

2.3 Instrumentation Sampling

To remove many of the existing limitations of instrumenting code online, we use the *full-duplication instrumentation sampling framework* [9], a fully automatic compiler transformation that transparently reduces the overhead of executing instrumented code with minimal impact on accuracy.

The key idea of the full-duplication technique, illustrated in Figure 1, is as follows: when a method is instrumented, the body of the method is duplicated and all instrumentation is inserted into the duplicated code. The original version of the method is only minimally instrumented at sample points that allow control to be transferred into the duplicated code. Samples are taken on regular intervals and control is transferred into the duplicated (and instrumented) code for a finite amount of time. A counter-based sampling mechanism is used to provide a flexible sample rate and deterministic sampling.

This technique has been shown to have low overhead and high accuracy for several examples of instrumentation [9]. This prior work focused on the design and implementation of the sampling technique in an offline setting without any feedback-directed optimizations. The technique is also flexible, allowing wide range of instrumentation techniques to be used without modification, making it easy to apply existing instrumentation techniques online.

The main disadvantage of the full-duplication technique is that the entire method is duplicated, increasing both space (by a factor of two) and compile time (by roughly 30% [9]).[3] This compile time increase affects bottom line performance when compilation occurs at runtime; therefore, candidates for online instrumentation must be chosen carefully.

[3]Additional compilation overhead due to full-duplication transformation is limited to 30% rather than 100% because the duplication occurs late in the optimization processes, as one of the last machine-independent optimizations [5].

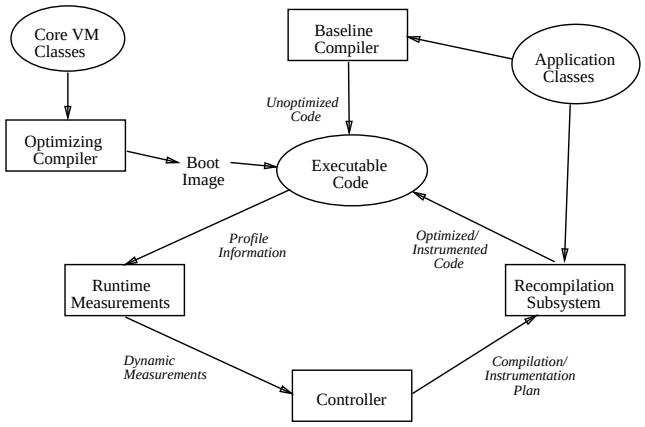

Figure 2: Overview of the general design of Jikes RVM adaptive optimization system [6]

2.4 The Jikes RVM Adaptive System

This section summarizes the Jikes RVM adaptive optimization system [6], which provides the base system for our work. The Jikes RVM [1] is a research virtual machine developed at IBM T.J. Watson Research Center. The system is written almost entirely in the Java programming language [2], and uses a compile-only approach (no interpreter). Figure 2 gives an overview of the general design of the Jikes RVM's adaptive optimization system. The architecture contains three main components: *runtime measurements*, the *controller*, and the *recompilation subsystem*. Methods are compiled with the non-optimizing *baseline* compiler upon their first execution, and an aggressive optimizing compiler is applied selectively by the adaptive optimization system.

Figure 3 presents the base implementation of the Jikes adaptive system (version 1.1b) that this work extended. It provides implementation details for the design given in Figure 2. In addition to the threads created by the application, the adaptive optimization system creates its own threads: three organizer threads, the controller, and a compilation thread. The overhead from organizers and controller is approximately 1%. Further details are provided below on the three components of the adaptive optimization system.

The runtime measurements subsystem performs profiling throughout execution using a low overhead timer-based sampling mechanism to identify frequently executed methods. Because this sampling mechanism occurs prior to a threadswitch in the Jikes RVM thread scheduler, no instrumentation of the executing method is required. Periodically, sampled methods are passed to the decision-making component, called the controller.

The controller uses a cost/benefit model to determine what action should be taken for each recompilation candidate it considers. When considering whether to optimize a particular method, the viable choices are to do nothing, or recompile at one of three optimization levels (O0, O1, O2) provided by the Jikes RVM's optimizing compiler. The goal of the controller is to provide good performance for both short- and long-running applications.

To determine whether to recompile a method, the controller first considers each optimization level, j, that is higher than the current optimization level, *cur*. The controller chooses the value of j that minimizes $C_j + T_j$, where C_j is the compilation cost at optimization level j and T_j is the expected future execution time of the method running at optimization level j. The controller next compares $T_j + C_j < T_{cur}$ to see if recompiling is better than leaving the method at its current level. If the controller finds such a j, it passes an optimization request to the recompilation system, which invokes the optimizing compiler on the method using optimization level j.

The model assumes 1) a method will execute for twice its current duration, 2) the method sample data determines how long a method has executed, and 3) the compilation cost and expected speedup are computed using offline averages as given in Table 1. The "FDO" line in this table will be explained in Section 3.3.

The Jikes RVM system also performs a simple form of feedback-directed optimization called *adaptive inlining* [6]. This technique uses the same coarse-grained timer-based sampling mechanism used to detect hot methods. Each sample is taken prior to a threadswitch and records the top two methods on the activation stack. These call edge samples are processed by the *inlining organizer* and decayed by the *decay organizer* to favor more recent samples. These samples are used in two ways. First, when a method is recompiled using the normal recompilation mechanism, the decayed samples are consulted to guide inlining decisions; more liberal code growth heuristics are used for determining whether to inline a sampled call site [6]. Second, if the inlining organizer detects a hot call edge in a method that is already optimized at the highest level (O2), it informs the controller of this fact, which can lead to the recompilation of that method (from O2 $\rightarrow$ O2) for the sole purpose of incorporating the new inlining decision.

The design of the adaptive optimization system also allows the controller to request that the optimizing compiler add instrumentation during optimization to ultimately drive feedback-directed optimizations. This feature, and the necessary modifications to the three components of the adaptive system, were not implemented in [6]. This implementation is the subject of this paper.

3. FDO IMPLEMENTATION

This section describes how the general adaptive optimization architecture described in [6] can be instantiated to perform online feedback-directed optimizations. Section 3.1 states our design goals. Section 3.2 describes how edge profiles are collected using the instrumentation sampling framework [9]. Section 3.3 describes how to enhance the Jikes RVM controller and related components to perform feedback-directed optimizations. Section 3.4 describes the four feedback-directed optimizations included in our system, including a novel algorithm for control flow graph node splitting.

3.1 Design Goals

Our online feedback-directed approach assumes an execution environment in which a lightweight mechanism is used to identify hot methods and promote them to higher levels of optimization. Our approach augments this by collecting fine-grained profiling information to provide additional information to the optimizing compiler for these hottest methods. Our goals for performing feedback-directed optimiza-

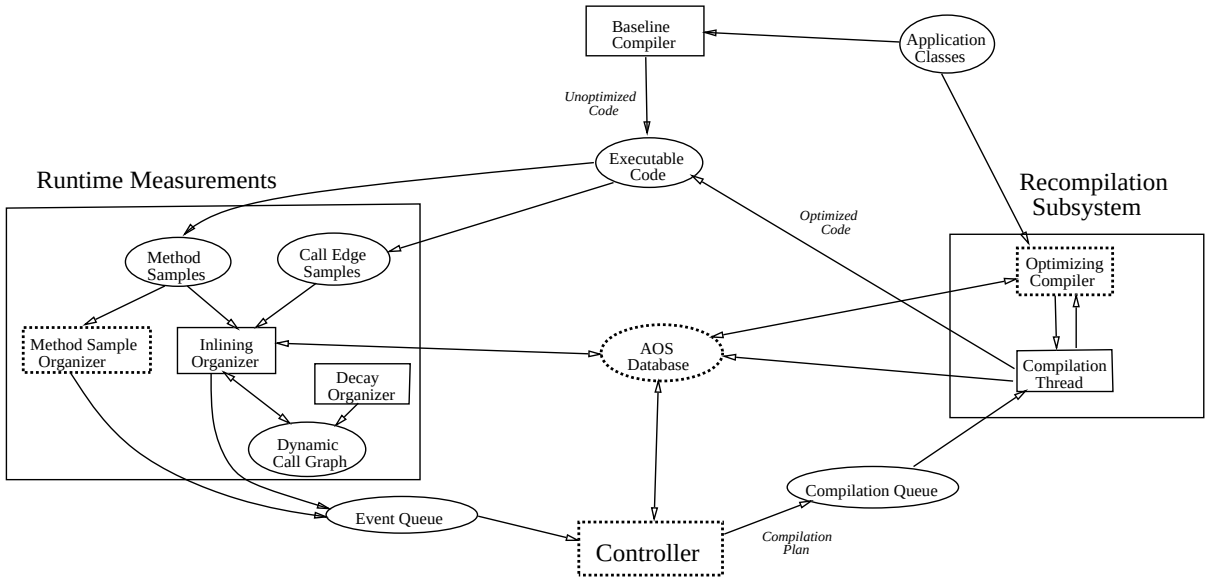

Figure 3: Default *implementation* of the Jikes RVM adaptive optimization system [6]. Components that were enhanced to perform online FDO are highlighted in dashed boxes.

tions are as follows, listed in decreasing order of importance.

1. Improve the performance of long-running applications Our primary goal is to reliably improve the peak performance of all long-running applications, regardless of their initial behavior or whether they exhibit phase shifts. Achieving peak performance for this range of applications requires a profiling mechanism that is flexible enough to support a wide range of feedback-directed optimizations, including those traditionally used offline. A second requirement is the ability to profile for more than just the first few moments of execution. It is unacceptable for an application to perform poorly for a long period of time (possibly several days) because the behavior observed during program startup was not representative of the entire execution.

2. Maintain startup performance Peak performance should be obtained without compromising the system's usefulness as a general VM that is able to provide high performance for both short- and long-running applications.

3. Improve performance as early as possible With all other factors constant, reaching high performance as early as possible is preferable. However, converging on high performance sooner is not worth compromising the level of performance that is eventually reached, or substantially degrading startup performance.

3.2 Intraprocedural Edge Profiles

The goal of intraprocedural edge counters [13] is to collect the execution frequencies of intraprocedural control flow edges. Execution frequencies of basic blocks can be easily derived from edge frequencies. Such profile information is useful for a variety of optimizations. It has been used *offline* in previous work for optimizations such as code reordering [40], instruction scheduling [31] and other classic code optimizations [18].

There are two subproblems involved in collecting edge profiles for the purpose of optimization: collecting the profiles, and making the profiles available to the client optimizations that use them. These two topics are discussed in the next two subsections, respectively.

Collecting Edge Profiles Previous work [13] has shown that careful placement of counters can reduce the overhead of collecting edge profiles. However, one advantage of the instrumentation sampling framework is that it significantly reduces the execution overhead of instrumentation. Therefore, to avoid unnecessary complexity, a simple counter placement is used. Optimal placement could still be used to reduce the counter space and code space, but is not necessary for the purpose of reducing time overhead.

A counter is placed before each conditional branch (to record the total number of times the branch was executed) and along the fall-through path (to record the not-taken frequency). The taken frequency is the difference between these collected counts.[4]

Using Edge Profiles Given the branch profiles collected using the technique described above, execution frequencies for all edges and basic blocks can be easily computed for the first phase of the optimization process. However, optimizations that alter control flow, such as loop unrolling, must adjust the profile to ensure that subsequent optimizations have accurate edge profiles. For most control-flow optimizations it is usually not difficult to determine a reasonable approximation for the weights after the transformation. Therefore, one possible solution is to modify all control-flow optimizations to approximate the new edge profiles. This solution is not desirable for three reasons.

First, the process of modifying control-flow optimizations is a time-consuming process. Every optimization must be inspected to see if it changes the control flow, and if so, up-

[4]Counting total executions and using it to compute the taken frequency was slightly more efficient than inserting an edge counter to explicitly count the taken frequency.

dated to propagate the edge profiles. Even relatively simple optimizations, such as branch optimizations and expansion of operators (such as casts and allocation), can introduce this problem. Second, modifying control-flow optimizations is prone to error. Mistakes can propagate through the optimization phases, potentially resulting in inaccurate profiles at the later phases. Third, when new optimizations are added, care must be taken to determine if they interfere with the propagation of edge profiles.

In the Jikes RVM, we use a technique that allows reasonable edge profiles to be available through all optimization phases, while minimizing the burden placed on the optimizations. This approach is similar to the Markov modeling of control flow described by Wagner et al. [48]. The key idea of the approach is to record the relative taken/not-taken ratio for all conditional branches rather than recording absolute edge frequencies. These ratios, or *branch probabilities*, are then used to compute the absolute edge frequencies on demand when needed by an optimization.

This approach places less burden on the optimizing compiler for two reasons. First, maintaining reasonable probabilities on all branches is a less burdensome task than maintaining the absolute frequencies of all edges. Second, if an optimization performs a transformation that creates a branch without a probability (or if the probability for a particular branch is missing from the profile data), the probability value can still be approximated using static heuristics [48]. Using this approach we were able to approximate reasonable edge profiles throughout compilation, while making only minimal changes to the optimizing compiler.

3.3 Online Strategy

This section describes the decision-making strategy for performing online feedback-directed optimizations. This strategy determines when, and on what methods, instrumentation and feedback-directed optimizations should be performed to minimize overhead while maximizing performance gains, ultimately trying to achieve the goals described in Section 3.1.

A high-level view of our online strategy is shown in Figure 4. Each circle represents a possible compilation state of a method. Each arrow (labeled 1–4) represents a transition (via recompilation) from one state to the next. Two profiling mechanisms are used during these transitions: the default timer-based sampling mechanism and the instrumentation sampling mechanism. Each transition is described below:

1. Execution begins with no instrumentation and the underlying adaptive system recompiles methods based on the collected timer-based samples using the cost/benefit model, possibly using multiple optimization levels. Avoiding instrumentation during the early executions of a method reduces the risk of degrading performance for short-running applications.

2. After a method has been optimized "statically" (at runtime, but without feedback-directed optimizations) it continues to be monitored using the timer-based sampling mechanism. If it remains sufficiently hot, it is instrumented to collect fine-grained profile information, such as control or data profiles, using the instrumentation sampling mechanism. Because the instrumented methods are known to be heavily executed, instrumentation sampling is needed to maintain competitive performance.

3. After the instrumented method has run long enough to collect sufficient profiling information, the method is recompiled again and feedback-directed optimizations are applied. Exactly how long the instrumented code should be executed depends on several factors, including the type of instrumentation being performed and the sample frequency of the instrumentation sampling. Speed of convergence is the lowest priority design goal (see Section 3.1) so longer instrumentation periods are preferable to ensure that accurate profiles are collected.

4. As time passes, program behavior may change. To ensure that performance does not degrade slowly over time as the profile becomes out of date, profiles can be periodically reevaluated. Reevaluation can be triggered by simple techniques, such as a timer mechanism, or by more advanced phase-shift detection algorithms [28, 23].

As described, our system employs a two-step approach to profiling. First, the system uses the existing Jikes RVM lightweight timer-based sampling mechanism to find hot methods [6]. Second, for the hottest methods, the system employs a heavier-weight instrumentation mechanism to gather more detailed profile information. This approach can be generalized to include a hierarchy of profiling mechanisms, where each subsequent technique provides more information at the cost of additional overhead, but is applied on a smaller portion of the application. This approach is consistent with the general design of the adaptive optimization system [6]. A similar approach was also used in [43]. In their work the first profiling mechanism used invocation and back-edge counters to determine hot methods. The second mechanism used instrumentation to capture a value or type profile for variables determined to be potentially important. These candidate variables are determined using an *impact analysis* [43], which is performed during an earlier compilation of the method.

Implementation Implementing this strategy requires some changes to the Jikes RVM adaptive system, the most important of which is introducing the notion of instrumentation and feedback-directed optimizations to the cost/benefit model. Figure 3 presents the base implementation of the Jikes adaptive system (version 1.1b) that this work extended. The components of the original adaptive optimization system that were enhanced to incorporate instrumentation and FDO are marked with dashed rectangles in Figure 3; these enhancements are discussed in the remainder of this section. Further details are provided below on the three components of the adaptive optimization system.

The runtime measurements subsystem contains an organizer thread, called the *method sample organizer*,[5] that processes timer-based method samples, aggregates them, and passes them to the controller via an event queue. In the base system this thread does not notify the controller about hot methods that are at the highest optimization level (O2)

[5]This component was called the *hot method organizer* in [6, 7].

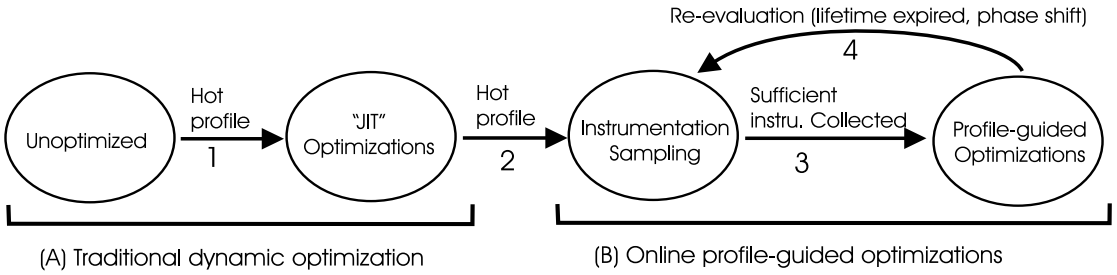

Figure 4: A graphical representation of our online strategy for performing profile-guided optimization. Each circle represents the possible state of a method. The transitions (labeled 1–4) are described in detail in Section 3.3. Transitions 1 and 2 are triggered using the timer-based sampling profiler. The counter-based profiler is used in the "Instrumentation Sampling" state.

because without FDO these methods are at their peak optimization level. The method sample organizer was modified to also pass O2 methods to the controller, so they could be considered for instrumentation and feedback-directed optimizations.

The cost/benefit model of the controller was modified to incorporate the notion of instrumentation and feedback-directed optimizations. When considering a method for optimization *without* FDO, the only options were to do nothing, or optimize at a higher optimization level. For FDO, a third choice is added to the model that consists of compiling with instrumentation inserted followed by recompiling with feedback-directed optimizations. Although instrumentation can be performed at any optimization level, in this work we chose to insert it at the highest optimization level (O2) to ensure that the effects of optimization are reflected in the profile.

The estimated cost of performing FDO consists of three parts: 1) the estimated cost of recompiling the method with instrumentation, 2) the estimated overhead of executing the instrumented code for the instrumentation duration, and 3) the estimated cost of compiling the method a second time to apply the FDOs. The expected benefit of the FDO choice is the estimated performance improvement that will occur after feedback-directed optimizations have been performed. The estimated speedup of all optimization levels (including FDO) is based on averages from offline performance measurements. The values used for this work are given in Table 1. The cost for the FDO option includes the three quantities mentioned above: the recompilation cost (at O2) to perform instrumentation, the overhead cost of executing the instrumented code, and the recompilation cost (at O2) for using the profile.

If the expected benefit of FDO outweighs the expected cost, the controller notifies the recompilation subsystem that the method should be recompiled with instrumentation inserted and the instrumentation sampling transformation applied. A sampling threshold value of 5,000 is used, i.e., every 5,000th check executed transfers control into the duplicated code containing the edge instrumentation, and one acyclic path is executed. This sample rate was chosen because it allows execution to continue at close to fully optimized speed [9] during the instrumentation period.

Once the instrumented method has executed for a sufficient duration, the collected edge profile can be examined so that feedback-directed optimizations can be applied. To

	Relative Compilation Cost	Relative Speedup (Benefit)
Baseline	1.00	1.00
Opt 0	37.83	4.21
Opt 1	80.81	5.84
Opt 2	223.99	6.01
FDO	671.97	6.48

Table 1: Estimated cost/benefit of optimization decisions relative to the baseline compiler. For example, Opt 0 is assumed to take 37.83 times longer than the baseline compiler, but produce code that will run 4.21 time faster.

make this possible, an "alarm clock" mechanism was added to the adaptive optimization system to allow the controller to reconsider instrumented methods after some duration. The alarm clock enables the association of a particular event with a "time" in the future; when the time arrives, the alarm clock sends the event to the controller for processing. Time in the Jikes RVM adaptive system is based on the number of time-based samples that have been taken, which is roughly 100/second. In this work, the alarm clock was set to 1000, which means the instrumented code was available for execution for approximately ten seconds. This value was chosen to avoid sampling for a short burst period.

Once the instrumented method is compiled, the new code is installed and an alarm is set with an "examine instrumentation" event; upon receiving this event, the controller examines the profile that has been collected for this method. The current implementation uses the simple strategy of testing whether the number of samples taken in this method is above a certain threshold (100 samples). If not, the alarm is reset and the instrumented method continues executing for another time period. We have yet to experiment with more advanced strategies; however, there are a number of possibilities. For example, the number of samples taken during the previous instrumentation period could be used to estimate the additional time needed to accumulate the desired number of samples.

When the controller decides that enough profiling information has been collected, the instrumented profile is stored in the *AOS database* and the method is scheduled for feedback-directed optimizations. Once this compilation

has completed, new calls to the method will transfer control to the highly-optimized FDO code. The current implementation does not employ a mechanism for triggering reevaluation of FDO methods.

The system described in this paper assumes that methods are not recompiled during instrumented profiling. As described in Section 2.4 the base Jikes RVM system will consider recompiling a method compiled at O2 if adaptive inlining can be applied, i.e., there exists a non-inlined hot call site in the method. Our version of the system temporarily disables this recompilation of an O2 method while an instrumented profile is being collected for that method. After the instrumented profiling completes, the normal adaptive inlining functionality resumes.

3.4 Feedback-Directed Optimizations

Many feedback-directed optimizations can benefit from edge profile information. Our system currently implements the following four feedback-directed optimizations: splitting, code motion, method inlining, and loop unrolling. Splitting requires control flow graph edge profiling; the other three optimizations use basic-block profiling. Recall that basic-block frequencies can easily be derived from edge frequencies. Each optimization is described below.

Feedback-Directed Splitting Splitting [17, 16] is a compiler transformation originally designed to reduce the overhead of message sends in the Self programming language. The goal of splitting is to expose optimization opportunities by specializing sections of code within a method. Specialization is achieved by performing tail-duplication of conditional control flow to eliminating control flow *merges* (or side entrances) where data flow information would have been lost.

Splitting is an enabling transformation; it exposes optimization opportunities making existing optimizations more effective, specifically, those optimizations that depend on forward data flow information, such as load elimination, array bounds elimination, devirtualization, etc.

The main limitation of splitting is that the potential space increase is exponential. Algorithms that use simple static heuristics to perform aggressive splitting, such as *eager splitting* [17, 16], have been explored, but were shown to cause excessive code expansion making them unusable in practice. Improved approaches, such as *reluctant splitting* [17, 16], delay splitting until later in the compilation process when data flow information identifies regions of code where splitting would be beneficial. Although more effective than eager splitting, this approach requires substantial compiler infrastructure and can still result in unacceptable code expansion.

Our approach is to develop a feedback-directed splitting algorithm, resulting in two advantages over previous approaches. First, splitting efforts can be focused on hot paths, allowing more aggressive splitting than would be possible with static heuristics, because space is not wasted duplicating cold code. Second, the algorithm is simple to implement in that it does not impact any other analysis, so no modifications are necessary to the optimizing compiler other than the splitting transformation itself (unlike reluctant splitting). By performing splitting early in the compilation process, optimization opportunities are automatically exposed to the optimizations that follow. However, this second advantage can also be a limitation. Unlike previous approaches that consider the impact of splitting on data flow analysis [3, 14], our splitting decisions are based purely on edge-profiling information until a threshold is reached. Thus, it may perform splitting that does not improve data flow information or may miss an opportunity to improve data flow information because a merge node was not hot enough.

The splitting algorithm, described in detail in Figure 5, attempts to split (and thus specialize) all heavily executed paths through the method, while avoiding duplication of infrequently executed code. It is a greedy algorithm that eliminates heavily executed control flow merges by duplicating the merge node and redirecting the hottest incoming edge to jump to the duplicated (and now specialized) node. A *merge node* is defined to be any basic block that has multiple incoming control flow edges, but is *not* a loop header, i.e., loop cloning [17, 16] is not performed.

Figure 6 shows an example of one iteration of the algorithm. Notice that after one merge node is split, two new merge nodes are created. The algorithm iterates, continuing to greedily split the hottest merge node until either 1) a space expansion bound (`EXPANSION_FACTOR`) is reached for the method, or 2) there are no more merge nodes that are above a minimum splitting execution threshold (`MIN_SPLIT_THRESHOLD`). In our implementation we used `EXPANSION_FACTOR` $= 3$ and `MIN_SPLIT_THRESHOLD` $= 50$ samples. These values were originally selected as "reasonable values" that would allow optimization to be performed while keeping space expansion within reason. Later, we experimented with different values, but found only minimal performance variations. The only concrete conclusion learned from tuning was that `mtrt's` improvements began to decline if `EXPANSION_FACTOR` dropped below 2.5.

Step 10 in Figure 5 requires further elaboration. After an edge is redirected from `hotMerge` to `dupNode` (step 9), several edge frequencies must be adjusted. If `hotMerge` has multiple outgoing edges (as shown in Figure 6) their frequencies cannot be known precisely because path-profiling information is not available. Therefore, the weights of outgoing edges are approximated using a *constant ratios* assumption [32]. The ratio of the outgoing edge frequencies is assumed to stay the same as the original outgoing ratio (in this case, 50–50). Using this ratio, the total incoming frequency of `hotMerge` and `dup` is distributed among their outgoing edges. This approach is essentially the same as the technique used to update edge frequencies in the presence of control-flow optimizations, as discussed in Section 3.2, but with one key difference. The splitting algorithm updates the edge frequencies after each iteration, thus for efficiency our splitting implementation performs the updates manually. The generic "update frequencies" routine could have been used; however, because the changes are always local to the last splitting step performed by the algorithm, manually updating the frequencies on the edges that changed is more efficient.

Feedback-Directed Method Inlining Using profiling to improve method inlining decisions can have a significant impact on the performance of object-oriented languages [8, 27]. The Jikes RVM has the ability to perform inlining based on static heuristics, as well as adaptive inlining based on a call-edge profile collected via time-based sampling as shown in Figure 3. Although the Jikes RVM's adaptive inlining shows performance improvements for many benchmarks, the

```
Initialize:

1. Initialize hotMergeNodes: an ordered set containing all merge nodes,
     sorted by execution frequency.
2. methodSize = getOriginalMethodSize();
3. maxMethodSize = methodSize * EXPANSION_FACTOR;
4. hotMerge = hotMergeNodes.getHottestNode();

Iterate:

5. while (methodSize < maxMethodSize &&
6.        hotMerge.frequency() > MIN_SPLIT_THRESHOLD) {
7.    dupNode = hotMerge.clone();
8.    methodSize += dupNode.getSize();
9.    Redirect the hottest incoming edge of hotMerge to jump to dupNode.
10.   Update edge frequencies.
11.   Update hotMergeNodes:
        Insert new merge nodes and re-sort nodes with modified frequencies.
12.   hotMerge = hotMergeNodes.getHottestNode();
   }
```

Figure 5: Feedback-directed splitting algorithm

sampled profile is fairly coarse-grained and thus may not provide enough information to completely determine all inlining decisions; therefore, static inlining heuristics are still used to make decisions for call sites that do not appear among the coarse-grained sampled edges.

Basic-block frequencies provide a finer granularity of profile information that can be used to improve inlining decisions. In our modified Jikes RVM system, nontrivial call sites that are known to be infrequently executed based on the basic-block frequencies are not inlined, even when the original static heuristics would have suggested inlining the call site.[6] Similarly, call sites that are frequently executed based on basic-block frequencies, but are not present in the existing time-based call-edge profile, are given higher priority for inlining by expanding the inlining size restrictions while processing those sites (and while processing any new code introduced by inlining at those sites).

Feedback-Directed Code Positioning Code positioning rearranges the ordering of the basic blocks with the goals of increasing locality, thus decreasing the number of instruction cache misses, and reducing the number of unconditional branches executed. The base Jikes RVM system performs code reordering based on static heuristics, by marking some blocks as cold, such as exception handlers, and moving these blocks to the bottom of the generated code.[7] Our modified version uses basic-block frequencies to drive the *top-down* code positioning algorithm described by Pettis and Hansen [40]. This algorithm first chooses the entry basic block. It continues placing the "best successor" to the last-placed block by choosing the highest frequency outgoing edge that leads to a block that has not yet been placed. When all successors of the last-placed have already been placed, the algorithm continues by selecting the block with the highest basic-block frequency. This process continues until all blocks are placed. To minimize further basic-block movement the code reordering phase is one of the last phases of the machine independent optimizations.

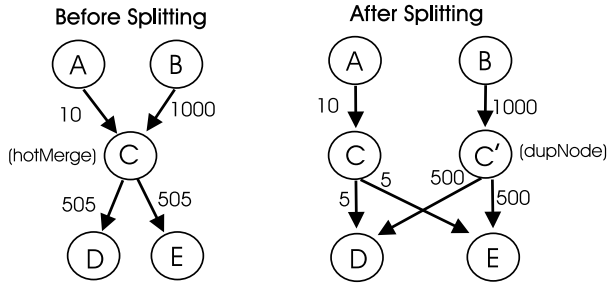

Figure 6: One iteration of feedback-directed splitting, where node `C` is the `hotMerge` node and `C'` is `dupNode`. After splitting, the estimated incoming edge frequency of `C` and `C'` is divided among their outgoing edges according to the ratio of the original outgoing edges of `C` (in this case, 50–50). This approach is similar to the technique used to update edge frequencies in the presence of control-flow optimizations, as discussed in Section 3.2.

Feedback-Directed Loop Unrolling Loop unrolling is a transformation in which the body of a loop is duplicated several times, allowing better optimization within the loop. The disadvantage to applying this optimization is that it can increase space substantially if it is not applied selectively. The base version of the Jikes RVM unrolls small (less than 100 instructions) inner loops four times. Our implementation uses basic-block frequencies to guide loop unrolling so that frequently executed loops are unrolled more to improve performance, but infrequently executed loops are unrolled less to preserve compile time and space. Our current implementation uses a simple heuristic to increase, or decrease,

[6] A trivial call site is a call to a method in which inlining would reduce code size.

[7] Recent versions of the Jikes RVM have improved these heuristics, as well as the mechanism for propagating this information along cold paths.

the loop unrolling factor based on the execution frequency of the loop header node; the loop unroll factor is doubled for hot loops (sampled more than 100 times), and halved for cold loops (sampled once or less).

4. EXPERIMENTAL EVALUATION

This section presents an experimental evaluation of our implementation of the techniques described in this paper. The implementation uses version 1.1b of the Jikes RVM. Although not a complete JVM, the performance of Jikes RVM has been shown to be competitive with that of commercial JVMs. The goal of these experiments is to validate the combined effectiveness of the instrumentation, feedback-directed optimizations, and the online strategy for applying them. The modified system is compared against the original Jikes RVM augmented with a more efficient implementation of guarded inlining [10]. This modified version offers higher performance than the original Jikes RVM; thus the baseline for all comparisons includes the guarded inlining enhancement.[8]

The experiments are divided into two sections. The first section describes a set of experiments using the SPECjvm98 [45] benchmark suite. The second section presents performance improvements using a more realistic server benchmark, SPECjbb2000 [46]. All experiments were performed on a 500 MHz 6 processor IBM RS/6000 Model S80 with 6 GB of RAM running IBM AIX 4.3.2. For all benchmarks, the Jikes RVM was run using 2 processors and a 400 MB heap.

4.1 SPECjvm98 Methodology

Understanding the performance impact of online feedback-directed optimizations can be difficult because of the number of factors that ultimately affect performance, including the overhead of instrumentation, the effectiveness of the feedback-directed optimizations, and even the behavior of the underlying adaptive optimization system. To gain a solid understanding of the performance impact (both overhead and improvement) of our online approach, long-running versions of the SPECjvm98 benchmarks were used. A standard "autorun" execution of the SPECjvm98 benchmarks consists of N executions of each benchmark in the same JVM; for these experiments, N was chosen separately for each benchmark to allow all benchmarks to run for approximately four minutes (using the size 100 inputs). This experimental setup ensured that each benchmark executed long enough for the underlying adaptive optimization system to approach a steady state, and also allowed the performance of each individual execution to be monitored, making it easier to identify where overhead was incurred and performance was gained. Details of this experimental setup are given in the first half of Table 2. The second column shows the number of runs for each benchmark, and the third column shows the best time achieved using the underlying base adaptive optimization system without our feedback-directed optimizations.

Because our FDO system is built on top of a non-deterministic adaptive system, it can be difficult to differentiate the impact of FDO from noise in the underlying system. To ensure confidence in our results, all timings reported were computed by taking the median of 10 timings. Although our methodology is relatively straightforward, the presence of multiple runs in the autorun sequence increases the potential for confusion; therefore, the timing computation is described in detail as follows. For each benchmark, the SPECjvm98 autorun sequence was executed to produce a dataset of N timings. This process was repeated 10 times to produce 10 such datasets. A *representative dataset* (or *representative autorun sequence*) was then computed by taking the pointwise median of the 10 datasets, i.e., run n of the representative dataset is computed by taking the median of run n across all 10 datasets. This representative dataset was then used as the timing data for the benchmark in question. Unless specified otherwise, text mentioning "run n" of a benchmark is always referring to run n of the representative autorun sequence. As shown later in this section, the variation in the timing data was relatively small in most cases.

Steady-State Performance Figure 7 characterizes the peak performance impact of the feedback-directed optimizations described in Section 3.4. Each benchmark was run as described in the previous paragraph with and without our online approach for feedback-directed optimizations.[9] Figure 7 compares the difference in *peak performance* of the two systems, where peak performance is defined as fastest run in the representative autorun sequence for that benchmark. The height of each bar represents the percent improvement achieved by performing feedback-directed optimizations. Improvements range from 0.9% (`javac`) to 16.9% (`mtrt`), with a geometric mean of 4.3%.

The patterns within each bar show how much each individual optimization contributes to the total performance win. The breakdown was computed by turning on each optimization, one at a time, starting with code reordering and working upward. This breakdown should be considered only a rough estimate of the individual contributions because there are many (mostly positive) interactions between the optimizations.

Code reordering provided reasonable speedups for several of the benchmarks. Splitting provides a large performance boost for `mtrt`, which makes heavy use of accessor methods, many of which are inlined using a runtime guard [22]. After splitting, many of these guards can be identified as redundant and eliminated.

Using edge profiles to improve inlining decisions showed

[8]The version of the Jikes RVM used in this work uses a single coarse-grained lock to coordinate accesses to underlying VM data structures. In particular, the lock is held during the entire compilation process, which can prevent compilation from occurring concurrently with the application when the application is directly or indirectly querying VM data structures. Because this coarse-grained synchronization is unnecessary, we experimented with (unsafely) eliminating the holding of the lock during compilation, which resulted in correct executions for all benchmarks. Because one benchmark (`javac`) showed increased overhead using the existing locking mechanism, we used the approach that does not hold the lock during compilation. More recent versions of the system (as of 2.1.1) no longer use a single lock, but instead employ a fine-grained locking mechanism for access to VM data structures.

[9]Recall that the base Jikes RVM system already performs loop unrolling, static and adaptive inlining, and code reordering. This graph shows the performance improvement of modifying these optimizations to take advantage of edge profile information.

Benchmarks	Application Characteristics		Compilation Statistics (with FDO)						Space Overhead
	# Runs	Best time	Total # compilations	Percent Breakdown					
				Base	O0	O1	O2	INST/FDO	% Increase
`compress`	11	18.8	382	93	2	3	1	1	6.3
`jess`	36	6.3	915	84	6	6	2	1	6.2
`db`	14	17.3	399	94	2	2	1	1	5.8
`javac`	20	10.9	1,575	70	16	32	1	1	4.6
`mpegaudio`	11	19.9	704	75	11	10	3	2	6.9
`mtrt`	54	4.1	634	78	8	10	2	1	6.6
`jack`	15	15.5	738	80	10	6	3	1	6.5
Geomean	19	11.5	787	82	6	7	2	1	6.0

Table 2: Recompilation statistics and space overhead for the SPECjvm98 benchmarks

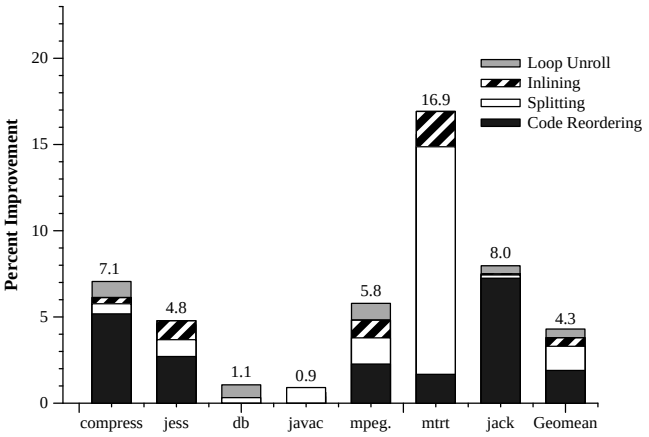

Figure 7: Peak performance improvement when using instrumentation and feedback-directed optimization

improvement for `jess`, `mpegaudio` and `mtrt`, even with the base system already performing adaptive inlining, demonstrating that the fine-grained profiling provided by instrumentation can be used to improve coarse-grained information. Loop unrolling provided little speedup for these benchmarks, showing only a slight improvement for `compress`, `db` and `mpegaudio`.

Overall, the speedups provided by these optimizations are reasonable, but by no means groundbreaking; however, the total performance improvement provided by these particular optimizations is not the main contribution of this paper, nor are these speedups meant to represent the maximum improvements possible from FDO. Instead, these optimization are being used mainly as proof-of-concept examples to help validate our online FDO infrastructure. The main goal of this work, as discussed next, is to demonstrate that our system makes it practical to perform these kinds of optimization *online*, and thus opens the door for developing new feedback-directed optimizations.

Online Performance To measure online adaptive performance, previous work has often reported simple metrics, such as the performance of the first and best runs [6, 43]. Unfortunately these simple metrics are not comprehensive regarding online performance because they disregard the performance of several runs, many of which may have incurred severe overhead. One alternative is to compare total execution time, rather than best time; however, this approach can also be misleading because short periods of high overhead can go unnoticed if the benchmark is configured to run long enough.

Figures 8 and 9 report the online performance of our system using a less forgiving metric — by comparing the performance of all runs. Figure 8 reports all execution times from the (representative) autorun sequences of both the FDO and non-FDO systems. Each run of the autorun sequence becomes a point in the graph,[10] with the point's placement determined as follows: for the point corresponding to run n, the y-coordinate is the execution time of run n (in seconds), and the x-coordinate is the cumulative time of the autorun sequence (the sum of the execution times of runs $1..n$). Points within the autorun sequence are connected by a line in the graph for easier viewing, but these lines have no formal significance.

As shown in Figure 8, both the FDO and non-FDO systems improve the performance of all benchmarks as the autorun sequence progresses, with the most substantial improvements being obtained relatively early in the sequence. However, the FDO system eventually achieves better performance. For several benchmarks FDO also decreased the total time needed for the autorun sequence to complete, i.e., the FDO-system completed all n runs in less time than the non-FDO system. This is most visible for benchmarks `mtrt` and `jack` in Figure 8, where the final (rightmost) point of the FDO curve occurs earlier in time (further to the left) than the final point from the non-FDO system.

To confirm that the improvement obtained by the FDO system is significant beyond the noise in the timings, an error bar appears on each point in the Figure 8. Recall that each point is the representative time computed as the median of 10 runs (as described in Section 4.1). The error bars represent the range within which 50% of the timings fell for that run. The error for most runs is small, often too small to be visible on the graph. The absolute magnitude of the error is not particularly relevant; more important is the observation is that the error is small relative to the performance gap between the FDO and non-FDO systems, confirming that the speedup due to FDO is significant beyond the noise.

Figure 9 is similar to Figure 8, but reports the relative performance difference between the two systems, rather than absolute timings. Each run from the FDO system was compared against the corresponding run from the non-FDO runs to create a point in the graph. The percent improvement due to FDO is the y-coordinate of each point (higher means

[10]To help reduce noise and make the graph more readable, benchmarks with individual execution times of less than ten seconds (`mtrt` and `jess`) had consecutive timings grouped into pairs of two; the sum of each pair was then used as a single timing.

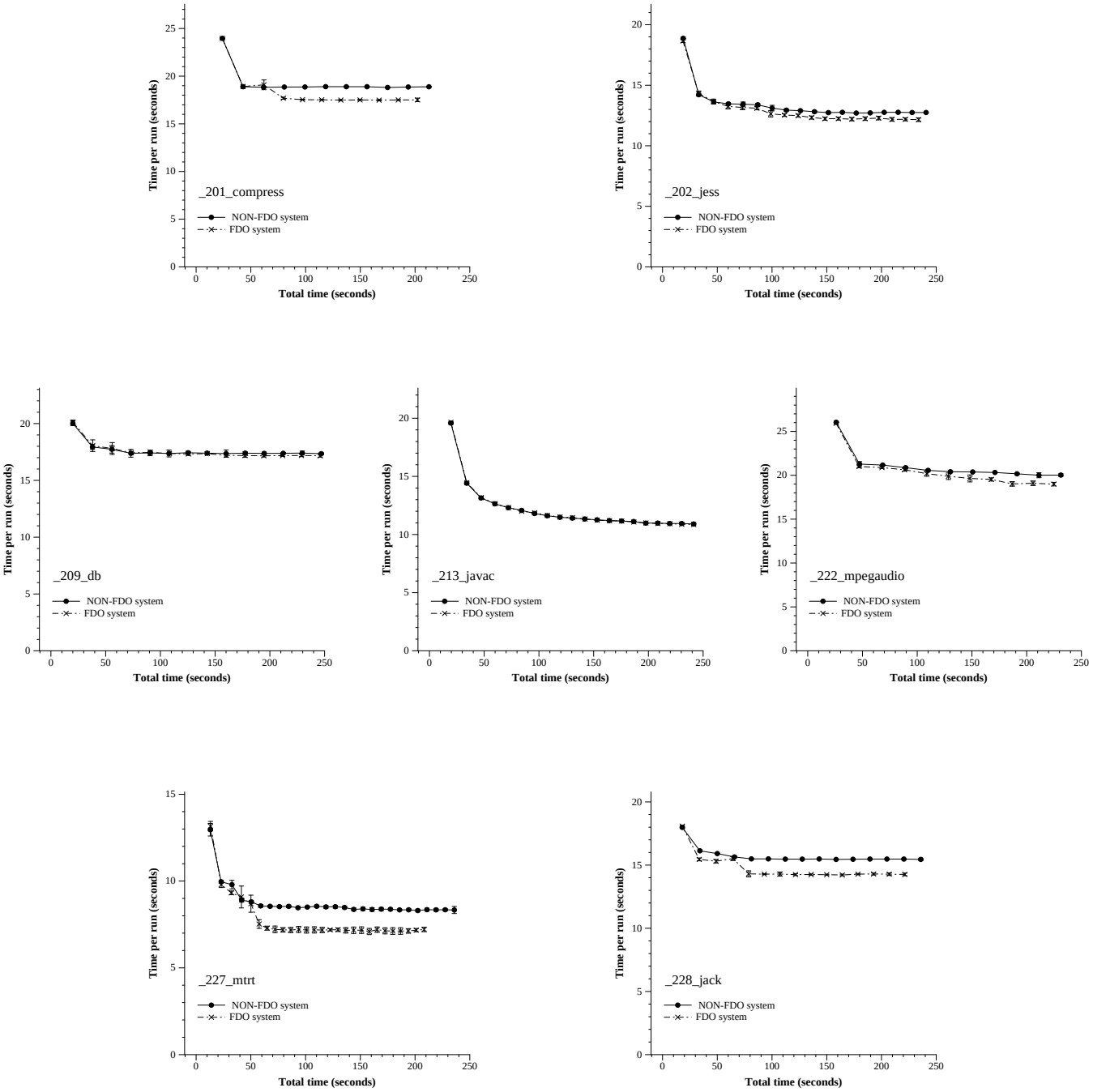

Figure 8: Online performance of both the base Jikes RVM adaptive system (non-FDO) and our FDO extension. Each point represents the execution time of a run within the representative autorun sequence. The error bar on each point represents the range within which 50% of the timings fell for that run. The error for most runs is small, often too small to be visible on the graph.

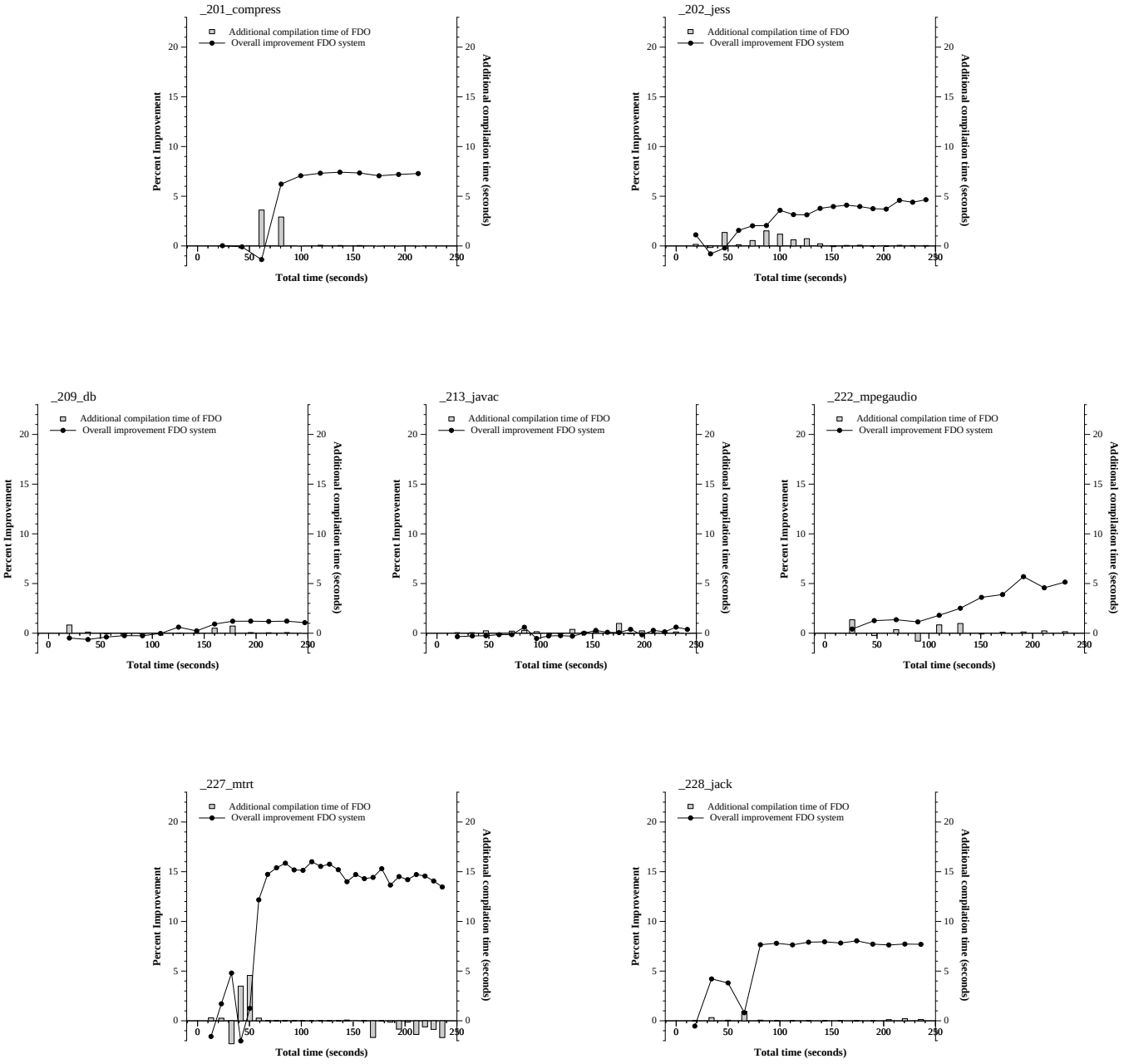

Figure 9: Online performance improvement when using instrumentation and feedback-directed optimizations (relative to the non-FDO Jikes RVM adaptive system). Each point represents the completion of a run of the benchmark. Each bar represents the additional compilation performed by the FDO system, relative to the non-FDO system

more improvement due to FDO), and the total execution time (of the non-FDO run) is the x-coordinate, showing the point in time at which this particular improvement was achieved. The graph reconfirms that FDO improved the performance of most of the executions, regardless of the increased overhead due to the additional recompilation and instrumentation. As the benchmarks run longer and more methods are instrumented, FDO continues to improve performance for most of the benchmarks. In particular, `mtrt`, `compress` and `jack` had the biggest speedups, approaching 17%, 8%, and 8%, respectively. The largest degradation for any run was less than 2%, demonstrating that the eventual improved performance of FDO does not come at the cost of severe overhead to any one execution.

The bars superimposed on the graph reports one aspect of the additional overhead being incurred by the FDO system: compile time. The height of the bar represents the additional compile time (measured in seconds) that is performed during that run by the FDO system, relative to the non-FDO system. Because the FDO system performs extra recompilation for instrumentation and FDO, there is generally an increase in compile time. These bars confirm that compilation overhead is responsible for some of the downward trends in the graphs, such as those that occur in `mtrt` and `compress`. However, it is possible for the FDO system to perform *less* compilation on any given run, resulting in a negative bar in Figure 9, as shown near the end of the runs for `mtrt`.[11]

Although benchmarks such as `mtrt` are important because they show the potential effectiveness of feedback-directed optimizations, benchmarks such as `db` and `javac` are equally important (if not *more* important) because they demonstrate the behavior of the system when the feedback-directed optimizations provide only minimal performance improvement. It is often the "worst case" benchmarks that are troublesome for systems performing online FDO; when the particular optimizations being applied do not provide speedups, the overall performance can be substantially degraded over the original non-FDO system, thus limiting the usability of FDO in practice. In our system, performance was never substantially degraded, even when the feedback-directed optimizations had only minimal impact. This is an important result because removing the performance risk of online FDO was one of the original goals of our profiling infrastructure; by minimizing the degradation in the worst cases, it makes it possible to achieve the performance gains in the best cases.

In addition, these results are particularly encouraging considering that much of the overhead being incurred is fixed cost overhead (such as the full-duplication compile-time increase and the runtime overhead of the checking code) that will not increase as more instrumentation and optimizations are added to the system.

[11]The addition of another choice (FDO) to the controller could cause less compilation to occur for two reasons. First, the FDO system could shift the compilation effort, performing more compilation in earlier runs and leaving less for later runs. Second, the FDO system could actually decrease the *total* amount of compilation that occurs. For example, there could be a method in the non-FDO system that is compiled three times, working its way through all optimization levels ($O0$, $O1$, and $O2$). This same method could be immediately selected for instrumentation and FDO by the FDO system, thus requiring only two compilations.

Space Overhead One concern of the approach described in this paper is the increase in space because of extra compilation. Additionally, instrumentation uses the full-duplication sampling technique, which doubles the size of the instrumented method. Further, three of the four feedback-directed optimizations have the potential to increase code size as well.

Fortunately, few methods required instrumentation to achieve the reported speedups. Columns 4–10 of Table 2 show the compilation statistics when FDO is performed. The first of these columns shows the total number of method compilations (baseline and optimized) that occurred; the next five columns report the percentage of compilations performed by each compiler and optimization level.[12] For all benchmarks, instrumentation and FDO was performed on only 1% or 2% of the compilations that occurred.

The final column of Table 2 shows the total space increase when using instrumentation and FDO. The version of the Jikes RVM used in this study does not garbage collect methods that are no longer being executed,[13] so space usage is computed by summing the machine code size of all compiled methods. The space increase ranges from 4.6% to 6.9%, with a geometric mean of 6.0%. This increase is reasonable given the speedups obtained, and is small compared to the space savings that could be obtained using other techniques, such as adding an interpreter to the system [43], or by garbage collecting dead methods.

4.2 Server Benchmark Performance

Although the SPECjvm98 benchmarks are useful for understanding online behavior, our long-term goal is to improve the performance of server applications. This section evaluates performance using the `SPECjbb2000` server benchmark [46], a Java application designed to evaluate server performance by emulating a 3-tier middleware system.

A standard compliant run of `SPECjbb2000` was used, which involves executing a series of 8 "points" in succession; each point executes a warm-up period for thirty seconds, followed by two minutes of timed execution. Execution begins with a single thread (during point 1) and an additional thread is added for each additional point. The performance of each timed segment is reported as a throughput (transactions per second).

Table 3 shows the performance of our FDO system (relative to the non-FDO Jikes RVM adaptive system) for `SPECjbb2000`. As shown by the figure, FDO steadily improves performance throughout 8 points, reaching a maximum improvement of 3.5%. The bar chart in Figure 10 shows a rough breakdown of the benefit from each of the four feedback-directed optimizations. Code reordering provides the majority of the speedup, with some additional benefit coming from splitting and inlining. Loop unrolling does not improve performance for this benchmark.

5. ADDITIONAL RELATED WORK

The instrumentation sampling technique used in this work was previously described in [9]. This prior work focused on

[12]Recall that all executing methods are first compiled with the baseline compiler, and may be subsequently compiled (possibly multiple times) with the optimizing compiler.

[13]This feature was recently added in version 2.0.2 of the Jikes RVM.

Point	Throughput		% Improvement
	Original	FDO	
1	4168	4168	0.0 %
2	8123	8187	0.8 %
3	7959	7966	0.1 %
4	7428	7621	2.6 %
5	7522	7695	2.3 %
6	7411	7640	3.1 %
7	7325	7574	3.4 %
8	7282	7536	3.5 %

Table 3: SPECjbb2000 server benchmark performance, with and without online feedback-directed optimization

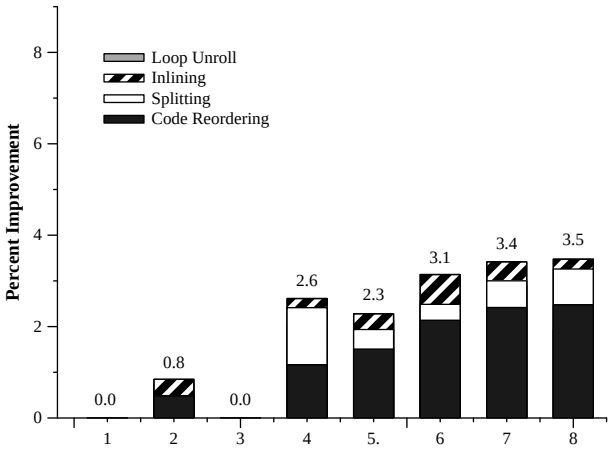

Figure 10: Performance improvement of FDO on the SPECjbb2000 server benchmark

the design and implementation of the sampling technique itself, and was evaluated *solely* in an offline setting. No online strategies using instrumentation sampling were presented or evaluated, and no feedback-directed optimizations were described. The basic architecture of the Jikes RVM adaptive optimization system was described in [6]. Although instrumentation and feedback-directed optimizations were included in the general architecture, a specific design and implementation of these features in a working adaptive system was not discussed. Our current work is an instantiation of these previous ideas into the Jikes RVM; our experiments with this online implementation use feedback-directed optimizations supported by instrumentation sampling to significantly improve the performance of Java programs. Thus, the previous work on instrumentation sampling and the previous description of an adaptive optimization system are the design foundation for the contribution of this work, namely, an investigation of feedback-directed optimizations in a VM using instrumentation sampling. All the engineering choices in the implementation and the experimental results obtained are new.

The Arnold-Ryder profiling framework [9] requires that execution remain in the duplicated code for a bounded amount of time. Although the implementation and evaluation in [9] focused on collecting profiles along acyclic paths in the duplicated code, the paper also suggested variants of this approach, such as staying in the duplicated code for multiple loop iterations. Hirzel and Chilimbi [29] implemented and evaluated this variation in the context of x86 binaries, calling it *Bursty Tracing*. In their system the instrumented method is modified so that upon taking a sample, execution remains in the duplicated code until a fixed number of checks are executed. This approach enables profiling for longer bursts, allowing more effective collection of temporal memory reference profiles.

Chilimbi and Hirzel [20] use this technique to collect memory reference profiles that are used to guide prefetching optimizations of x86 binaries in a fully automatic online system. Their technique leads to overall execution time improvements of 5–19% for several of the memory-performance-limited benchmarks of the SPECint2000 suite [20]. Although both online systems strive to achieve the same basic goal – performing instrumentation and optimization online – the implementation details of the two systems differ substantially. Their system uses the Vulcan binary rewriting tool [42] to transform x86 binaries into self-optimizing programs; therefore, it has the advantage of being language independent.

Their full-duplication profiling transformation is applied offline, prior to execution, and thus avoids the overhead of performing the transformation at runtime; however, by performing the transformation on all methods prior to execution, the size of the program is increased by at least a factor of 2. After profiling for some period, a subset of the procedures are optimized by using Vulcan to dynamically clone and modify the code.

A detailed comparison of the effectiveness of the two systems is difficult to make for several reasons. First, there are many fundamental differences between the two systems, including the profiling information collected, the optimizations performed, and even the language on which the systems operate. Second, their results are reported as "total

time" improvements[14] making it hard to evaluate the pause times introduced by the instrumentation and optimization infrastructure. Finally, the study focused on the memory-intensive benchmarks of the SPECint2000 suite; thus the potential negative impact of their system for benchmarks that do not benefit from their prefetching optimization is unclear. The exact space requirements of their system are not reported.

The IBM DK 1.3.0 [43, 44] is the most robust publicly described JVM that performs online instrumentation of optimized code. The work described in [43] uses code patching to profile values of method parameters. This idea is extended in [44] to capture dynamic call graph information to drive inlining. The results in these papers do not report the performance while executing the instrumentation; only the final "steady state" performance is given.

The MRL VM [21] uses a two-stage compile-only approach. Recompilation is triggered using two mechanisms. The first mechanism, similar to [30, 39, 43], inserts counters in the non-optimized code that track method invocations and the execution of loop back edges. These counters are initialized to 1,000 and 10,000, respectively, and are decremented at method entries and loop back edges. When the value of a counter becomes zero, the associated method is optimized by the executing thread. The authors mention, as discussed in Section 2.2, that choosing an appropriate threshold for all programs is challenging. The second mechanism uses a separate thread to scan different counters looking for recompilation candidates. These counters are incremented by the non-optimized code in the same manner as the others are decremented, and periodically reset by the separate thread. When the separate thread finds a counter above a threshold, it optimizes the method in parallel with the executing application. As described, the MRL VM does not perform any online feedback-directed optimizations.

Dynamo [12] is a binary translator that uses a technique called *next executing tail* (*NET*) [25] to approximate hot paths; their experimental results argue that traditional path instrumentation is unnecessary for online use. Although this may be true given the nature of Dynamo, our use of edge profiling (as well as several of the systems described in Section 2.2) is fundamentally different. Dynamo uses hot-path information not only for performing code reordering, but also for identifying code that needs to move from interpreted to optimized state; therefore, Dynamo emphasizes the speed of the profiling technique over its accuracy. Our online approach, however, assumes an execution environment in which a lighter-weight mechanism is used to identify hot sections of code and promotes them to higher levels of optimization. Fine-grained profiling information is used only to improve the decisions made by the optimizing compiler; therefore, the opportunity cost during the profiling period is smaller. As a result, it becomes a worthwhile tradeoff for systems like ours to profile for a longer period of time to gain the benefits of a more accurate profile.

Traub et al. [47] describes *ephemeral instrumentation*, a technique that uses code patching to reduce the overhead of collecting edge profiles. The authors show that after a post-processing pass, such a profile can be used to guide superblock scheduling, producing speedups similar to those from a perfect profile, although for a few benchmarks the sampled profile was significantly less effective. As discussed in Section 2.2, it is not clear whether this technique can be used to perform general instrumentation.

Our feedback-directed splitting is similar to superblock scheduling [18], which eliminates side entrances along hot paths by performing tail-duplication. The key difference between our splitting algorithm and superblock scheduling is that the superblock formation algorithm duplicates each basic block at most once, and thus is most effective when there is only one hot path through the code. The potential for performing additional duplication after initial superblock formation is mentioned [18], but not fully described. Our splitting algorithm repeatedly selects the hottest basic block for splitting regardless of whether it has already been duplicated, and thus allows full duplication of multiple hot paths through a method.

Ammons and Larus [3] and Melski [36] use offline path profiles to guide code duplication aimed at improving the precision of data flow analysis, and thus potentially facilitating path-specialized optimizations. Subsequently, the results of the data flow analysis are used to eliminate those duplicated nodes/edges that do not improve analysis precision. Our feedback-directed splitting algorithm also aims at directly enabling code specialization through code duplication, but, in contrast, is an *online* technique. Our method duplicates code along hot edges after gathering profile information, focusing on paths containing merged subpaths of heavy frequency. This step is followed by analysis and transformation passes of the optimizing compiler to specialize the code along these newly exposed hot paths.

Bodik et al. [14] use offline edge or path profiles to guide code restructuring and duplication for partial redundancy elimination (PRE) and code motion. The emphasis in this work is on the integration of code motion and code restructuring to allow partial redundancy elimination with controlled code duplication. Several PRE algorithms are empirically compared. In comparison, our splitting algorithm (which is a code restructuring) is not specific to a particular analysis or optimization.

Diniz and Rinard [24] describe a dynamic-feedback system that automatically chooses the best synchronization policy for a program at runtime. The system alternates between sampling the environment in order to choose from among several code translations to use, and then executing the chosen translation for a production interval, before sampling again to see if another change is needed. This approach to adaptive compilation was used in a parallelizing compiler for object-oriented languages. Although the sampling is performed online in this system, it drives a choice between precompiled optimized versions of code, rather than driving runtime optimizations of the program.

Ayers et al. [11] used *offline* basic-block profiling information to guide inlining decisions. The highest priority is given to high-frequency call sites. Likewise, call sites that are in blocks that execute less frequently than the entry block are penalized. Our online feedback-directed inlining also shares these properties.

6. CONCLUSIONS AND FUTURE WORK

This paper describes the implementation and evaluation of a fully automatic online system for performing feedback-directed optimizations. The system consists of three components: a low-overhead edge count profiler based on instru-

[14]The execution times of the benchmarks are not specified.

mentation sampling, four feedback-directed optimizations, including a novel profile-guided splitting algorithm, and an online strategy for performing both the instrumentation and the optimizations. The experimental results show peak performance can be substantially improved; speedups ranged from 0.9% to 16.9% (averaging 4.3%) for SPECjvm98 and 8.9% for SPECjbb2000. The overhead of our technique is also encouraging. No individual execution degraded more than 2%, and much of this overhead is fixed-cost overhead that will not increase as more instrumentation and feedback-directed optimizations are added to the system.

Our future plans are to use the online infrastructure presented here to explore additional instrumentation techniques and feedback-directed optimizations. Current possibilities include method-level specialization based on value profiles, as well as data locality optimizations based on memory reference profiles. We also plan to move our implementation into the open-source release of Jikes RVM as well as continue exploring additional server applications.

7. ACKNOWLEDGMENTS

We would like to thank IBM Research and the entire Jikes RVM team for providing the infrastructure to make this research possible. We also thank Stephen Fink, David Grove, and Peter Sweeney for the helpful discussions about adaptive optimization. We are grateful to Martin Trapp for supplying the loop unrolling implementation. Rastisav Bodik, Stephen Fink, David Grove, Martin Hirzel, Peter Sweeney, and Laureen Treacy provided valuable feedback on an earlier draft of this paper. Finally, we thank Vivek Sarkar for his support of this work. Work by Rutgers researchers described here was supported, in part, by NSF CCR-9900988.

8. REFERENCES

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

[2] Bowen Alpern, Dick Attanasio, John J. Barton, Anthony Cocchi, Derek Lieber, Stephen Smith, and Ton Ngo. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 314–324, October 1999.

[3] Glenn Ammons and James R. Larus. Improving data-flow analysis with path profiles. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 72–84, June 1998.

[4] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *16th ACM Symposium on Operating Systems Principles*, October 1997.

[5] Matthew Arnold. Online instrumentation and feedback-directed optimization of Java. Technical Report DCS-TR-469, Rutgers University, December 2001.

[6] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 2000.

[7] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM: The controller's analytical model. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, December 2000.

[8] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F. Sweeney. A comparative study of static and dynamic heuristics for inlining. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, January 2000.

[9] Matthew Arnold and Barbara Ryder. A framework for reducing the cost of instrumented code. In *SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 168–179, June 2001.

[10] Matthew Arnold and Barbara G. Ryder. Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading. In *16th European Conference on Object-Oriented Programming*, June 2002.

[11] Andrew Ayers, Robert Gottlieb, and Richard Schooler. Aggressive inlining. In *SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 134–145, June 1997.

[12] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[13] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[14] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Complete removal of redundant expressions. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, June 1998.

[15] Mike Burrows, Ulfar Erlingson, Shun-Tak A. Leung, Mark T Vandevoorde, Carl A. Waldspurger, Kefvin Walker, and William E. Weihl. Efficient and flexible value sampling. In *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[16] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Comiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, 1992.

[17] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–15, November 1991.

[18] Pohua P. Chang, Scott A. Mahlke, and Wen-Mei W. Hwu. Using profile information to assist classic code optimizations. *Software—Practice and Experience*, 21(12):1301–1321, December 1991.

[19] Trishul Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *SIGPLAN 2001 Conference on Programming Language Design and Implementation*, June 2001.

[20] Trishul Chilimbi and Martin Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 199–209, June 2002.

[21] Michal Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[22] David Detlefs and Ole Agesen. Inlining of virtual methods. In *13th European Conference on Object-Oriented Programming*, June 1999.

[23] Ashutosh S. Dhodapkar and James E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *29th International Symposium on Computer Architecture*, 2002.

[24] Pedro C. Diniz and Martin C. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 71–84, June 1997.

[25] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: Less is more. In *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[26] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. An evaluation of staged run-time optimizations in DyC. In *SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 293–304, May 1999.

[27] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-guided receiver class prediction. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 108–123, October 1995.

[28] Michael Hind, V.T. Rajan, and Peter F. Sweeney. Online phase detection. Submitted for publication.

[29] Martin Hirzel and Trishul Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, pages 117–126, December 2001.

[30] Urs Hölzle and David Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Languages and Systems*, 18(4):355–400, July 1996.

[31] Wen-mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. In *The Journal of Supercomputing*, pages 229–248. Kluwer Academic Publishers, 1993.

[32] O. Kaser and C. R. Ramakrishnan. Evaluating inlining techniques. *Computer Languages*, 24(2):55–72, July 1998.

[33] Thomas P. Kistler. *Continuous Program Optimization*. PhD thesis, University of California, Irvine, 1999.

[34] Mark Leone and Peter Lee. Dynamic specialization in the Fabius system. *ACM Computing Surveys*, 30(3es):1–5, September 1998. Article 23.

[35] Renaud Marlet, Charles Consel, and Philippe Boinot. Efficient incremental run-time specialization for free. In *SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 281–292, May 1999.

[36] David Gordon Melski. *Interprocedural Path Profiling and the Interprocedural Express-Lane Transformation*. PhD thesis, University of Wisconsin, February 2002.

[37] M. Mock, C. Chambers, and S. Eggers. Calpa: A tool for automating selective dynamic compilation. In *Proceedings of the 33th International Symposium on Microarchitecture, pp. 291–302*, December 2000.

[38] Takeshi Ogasawara, Hideaki Komatsu, and Toshio Nakatani. A study of exception handling and its dynamic optimization in Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 83–95, October 2001.

[39] M. Paleczny, C. Vic, and C Click. The Java Hotspot(TM) server compiler. In *USENIX Java Virtual Machine Research and Technology Symposium*, April 2001.

[40] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *SIGPLAN '90 Conference on Programming Language Design and Implementation*, June 1990.

[41] S. Subramanya Sastry, Rastislav Bodik, and James E. Smith. Rapid profiling via stratified sampling. In *28th International Symposium on Computer Architecture*, 2001.

[42] Amitabh Srivastava, Andrew Edwards, and Hoi Vo. Vulcan: Binary transformation in distributed environment. Technical Report MSR-TR-2001-50, Micorosft Research, April 2001.

[43] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 180–195, October 2001.

[44] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. An empirical study of method inlining for a Java just-in-time compiler. In *USENIX 2nd Java Virtual Machine Research and Technology Symposium (JVM'02)*, August 2002.

[45] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98, 1998.

[46] The Standard Performance Evaluation Corporation. SPEC JBB 2000. http://www.spec.org/osg/jbb2000, 2000.

[47] Omri Traub, Stuart Schechter, and Michael D. Smith. Ephemeral instrumentation for lightweight program profiling. Technical report, Harvard University, 2000.

[48] Tim A. Wagner, Vance Maverick, Susan L. Graham, and Michael A. Harrison. Accurate static estimators for program optimization. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 85–96, June 1994.

[49] John Whaley. Partial method compilation using dynamic profile information. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 166–179, October 2001.